SUPER

VISUAL INTERFACES FOR OBJECT+RELATIONSHIP DATA MODELS

Yves DENNEBOUY
Martin ANDERSSON
Annamaria AUDDINO
Yann DUPONT
Edi FONTANA
Massimo GENTILE
Stefano SPACCAPIETRA

Swiss Federal Institute of Technology
EPFL-DI-LBD CH 1015 Lausanne
firstname.name@di.epfl.ch

ABSTRACT

SUPER is an exploratory project into the next generation of user-DBMS interfaces. Its main objective is to demonstrate that a visual paradigm can lead to powerful and user-friendly interfaces supporting all phases of the database life cycle (i.e., creation, manipulation and evolution). Visual interaction in SUPER is based on direct manipulation of objects and functions, with a special focus on providing users with maximum flexibility and independence from database technicalities. The set of tools offers facilities to meet the varied demands from categories of users with different levels of skill. Diagrammatic representations and a basic set of functions are better suited for novice and occasional users, while menus and dialog boxes speed up the dialog for expert users. At the same time, a consistent interaction style over the various functions and tools has been emphasized. SUPER has been designed as a front-end to a relational or an object-oriented DBMS, i.e., the persistence of data, consistency and concurrency problems, are delegated to an off-the-shelf database management system. The current prototype supports schema definition, query formulation and browsing, using a powerful data model based on objects and relationships.

1. INTRODUCTION

The first DBMSs (based on CODASYL) were designed for programmers. Their users were supposed to understand data storage structures and to be able to control what exactly happened at the level of data access. At that time (1969), the notion of interactivity did not even exist. Next, the relational model was invented for the benefit of non programmers. Its data structure is simple, it is easy to understand, and it hides all implementation issues; simple operations can be expressed in an assertional rather than procedural fashion. This led to the development of query languages, at first formal (algebra, calculus) and later user-oriented (SQL, QBE, QUEL). These languages made possible to access data in a question-answer mode, which naturally leads to interactivity.

No visual formalism has been proposed to represent a relational schema, as the relational structure is flat and relations have no links between them. On the contrary, there was a rapid growth of visual interfaces for Data Manipulation Languages (DMLs), notably FORAL [1] and especially QBE [2], the latter based on the paradigm of query formulation by visualization of an example.

Visual interfaces became a must after hardware/software solutions made them both possible and easy to program, and also because a significant change in the user profile occurred when DBMS products moved into microcomputers. The direct manipulation paradigm [3] soon emerged as a de facto standard.

According to [4], a **visual language** implies the systematic use of visual expressions to convey a meaning in a formal way. Subsets of these are **iconic** languages, which make extensive use of icons and images, and **graphical** languages, which primarily adopt diagrams. Most of the existing prototypes are based on graphical languages, as the users interaction with the DBMS is built on the diagrammatic representation of the database schema. Some prototypes can be classified as visual in a more general sense as they employ other representations, e.g. forms [5]. Almost all products and prototypes use ER (Entity-Relationship) diagrams for data definition but assume the relational algebra for data manipulation. This paper presents a visual language for data manipulation based on a data model with objects and relationships and a corresponding algebra [6]. The language has been developed as part of a set of visual tools, called SUPER, which are intended to offer a DBMS-independent interface to users. All the tools rely on the same data model.

Section 2 presents a short overview of current approaches to database visual manipulation. Our underlying data model and algebraic operators are briefly introduced in section 3. The two major sections discuss query formulation (section 4) and browsing (section 5). Section 6 reports on the implementation of the prototype.

2. VISUAL DATA MANIPULATION

The overview in this section focuses on the functionalities, rather than on quoting all existing propositions and tools. More information can be found in a survey by Batini et al. [4]. The taxonomy below mainly distinguishes browsers, assertional query editors (which we examine in more detail) and procedural query editors.

2.1. Browsers

Navigating in a database is an easy and natural way of looking at data when the schema consists of a set of interrelated objects, as in ER and OO (object-oriented) databases. Existing browsers use different presentation techniques. Some display objects in forms, others in a graphical format (e.g. ER diagrams), another uses a different icon for each object class. Some tools are solely intended for data browsing, as [7], ZOO [8], SICON [9], OdeView [10], MOBIAS [11], [12]. In some other tools, browsing is one of the activities that can be done through the visual interface. Examples are SNAP [13], Iris [14, 15], Pasta-3 [16], OOQBE [17], GOOD [18].

OdeView is a good example of a graphical browser for an object-oriented database system (Ode). The user is provided with a directed acyclic graph depicting the class hierarchy: The node corresponding to a class can be selected and its information be displayed in textual form. To browse objects, the user can start by visualizing the class: Ode objects can be displayed both in textual and pictorial form. References to other objects are displayed, which allows to navigate from one class to a related one. This kind of tool can be expected to become a standard for OO DBMSs.

2.2. Assertional query editors

There are several approaches to assertional query formulation:
- syntactic editing, which is a visual aid for textual query formulation;
- formulation by example, which deduces the query to be generated from the definition of an example of what the user wishes to obtain;
- graphical editors, which aid the query formulation by presenting the database diagram; operations can be specified either through menus or by visual techniques.

2.2.1. Syntactic editing

[5] proposes a visual aid to query formulation through syntax diagrams displayed together with the ER diagram. The query is built by binding syntax diagram elements to database schema elements. The query being constructed is also displayed in a textual form, which can be directly edited by the user. IQL [19] is a more recent tool that combines syntactic editing with graphical formulation of SQL queries. A menu lists all available operations (project, select, join, group by, …) that can be performed on visual representations of relations imported into a workspace. The user designates attributes involved in projections and selections by clicking on them; simple predicates are represented as an arc labeled with a relational operator connecting the attribute name with the value. It is not clear how complex predicates can be defined.

2.2.2. Formulation by example

This was the first technique to be adopted for visual query formulation on a relational database. QBE [2] is based on the display of the schema of the relations that the user wants to manipulate, with the rows and the columns needed to represent the tuples to be defined. The user fills in the appropriate columns to indicate the result fields, the values for selection conditions, and the join conditions. The interface is easy to use for simple queries (selection, projection, join), while more complex queries require more learning and additional mechanisms (e.g., condition boxes). CUPID [20] is an example of a more graphical approach: a join between two relations is performed by drawing a link between them. Formulation by example is still used in many prototypes [21].

2.2.3. Graphical editors

These editors are based on a semantic data model to visualize the database schema. As the diagram is displayed, it is easy to designate the elements involved in the query. Query formulation can be based on menus, or rather on a visual specification of operations. Extraction of the query sub-schema is usually done through a "point-and-click" mechanism. Some interfaces provide additional mechanisms, such as the automatic selection mode used by gql/ER [22] (the system determines the most likely path connecting the selected items); QBD*[23] uses a similar technique for query sub-schema selection, as well as for the construction of logical expressions involving entity attributes (i.e., predicates).
Some interfaces transform the query sub-schema into a structure more suitable for query specification. In [24] the user selects the entity type of primary interest (called "root entity set"), so that the query sub-schema can be transformed into a hierarchical structure; cycles in the sub-schema are automatically broken in the transformation. QBD* allows the simplification of the query sub-schema by using a transformation language. [25] follows a different approach: at each step an operator transforms a sub-schema into another sub-schema.
Tools differ in the formulation of predicates on the (possibly transformed) query sub-schema. Some interfaces [26, 24] provide predicate formulation via menus; in gql/ER QBE-like forms are used to specify conditions on the selected nodes. Only a few interfaces allow the use of a graphical formalism for expressing predicates, e.g., Pasta-3 or SNAP. This phase is followed by (and sometimes interleaved with) the selection of the information to be output from the query. This information can be selected directly by clicking on the query sub-schema, as in SNAP, or designated through the same structure used for the formulation of predicates (gql/ER, Pasta-3).
The last phase is the display of query results. Usually, occurrences are displayed in a tabular form, as in QBD*. SNAP provides the user with the choice between this format and a NF2 format, where

occurrences are arranged into "buckets"; the NF2 format is adopted also by GOOD. GUIDE [27] allows the user to choose among different formats for the display of query results.

2.3. Procedural query editors

Only few of the prototypes presented in the literature are based on procedural formulation. This approach is favored by Berztiss [28], who states that, generally speaking, humans prefer procedural approaches to non-procedural ones. Queries are expressed by designating schema elements in the diagram and by choosing operators in a palette or a menu. Alternating between operators and operands follows the logic of textual writing of the query. To simplify the writing of procedural queries Motro [29] proposes techniques for (partial or total) reuse of queries

2.4. SUPER

SUPER belongs to the assertional query editors family. It is a visual editor, where the term visual is intended to stress that emphasis is put on all aspects of visualization, not just on the diagrammatic representation of a database schema. For instance, the usage of menus and buttons has been minimized: it is mostly confined to file management commands (save, open, …). Direct manipulation of objects in the workspace is our first class technique for user interactions.
SUPER is comparable to the few other visual prototypes supporting query formulation (e.g., QBD*, Pasta-3, Graqula[30]). With respect to these competitors, the major advantage we see in SUPER is that it provides all of the following features:

- the query interface is consistent with the data definition interface, i.e. they interact with users based on a common set of data modeling concepts. This contrasts with what is, in our opinion, the major drawback of existing protoypes for visual querying, which support entity-relationship diagrams to display the database schema while query evaluation results in the display of tabular data in first normal relational form.
- query formulation interactions have been designed to obey visual criteria (readability, intuitive understanding, …) without any attempt to mimic operations of the underlying data manipulation language. In particular, there is no menu with DML operations.
- a higher degree of software engineering ideas have been incorporated, namely flexibility, reusability and backtracking. Flexibility is mostly achieved by freeing users from predifined patterns for interaction. Simply stated, users can do what they want, the tool adjusts to what they decided to do. For instance, uncomplete specifications for a new database schema can be stored and resumed when the user decides to continue the definition task. Flexibility is also achieved by providing alternative interactions to adjust to user's level of expertise. Reusability has been stressed to allow for reuse of schema specifications, queries and data. Backtracking encompassed error recovery techniques, which in SUPER allow users to correct errors directly where the error appears while preserving the work which has already been done. For instance, if the frame of the query (see section 4.1.2) is not what the user needs, the user can change it by direct manipulation and SUPER automatically corrects the existing structure of the query result and the predicates.
- data structures and operations are based on a data model supporting complex objects, explicit relationships and multi-instantion links. We believe this type of data model will emerge as a standard in the near future. Indeed, it represents an evolution of the current standard for CASE tools (the ER model) to include object-oriented features. Relying on such a data model has a fundamental, positive influence on the query formulation process and the way it can be visually expressed.

3. THE ERC+ DATA MODEL

The SUPER tools are built on the ERC+ data model which is an object based extension of the entity-relationship model, specifically designed to support complex objects and object identity. It is more a conceptual than an object-oriented model, as it models relationships among objects independently of their implementation. In short, it belongs to the new family of objects+relationships data models ([31], [32], [6]). ERC+ has the following features:

- **object types** bear any number of attributes;

- **relationship types** may connect any number of participating object types, and may have attributes; cyclic relationship types are allowed;
- a **role** name is associated to each object type participation in a relationship type. The participation is characterized by its minimum and maximum cardinalities;
- **attributes** may be either atomic (non decomposable) or complex, i.e. decomposable into a set of component attributes, which may in turn be either atomic or complex; an attribute is also characterized by its minimum and maximum cardinalities (mandatory/optional, monovalued/multivalued);
- two varieties of **multi-instantiation** links are supported, the classical "is-a" generalization and an additional "may-be-a" link. The latter states that there may be a non empty intersection between the populations of the related object types.

Figure 1 shows a simple ERC+ diagram: a single continuous line is used to represent a 1:1 link (mandatory monovalued), a single dotted line represents a 0:1 link (optional monovalued), a double dotted line represents a 0:n link (optional multivalued), a double line (once dotted, once continuous) represents a 1:n link (mandatory multivalued). Multi-instantiations are not shown in this example.

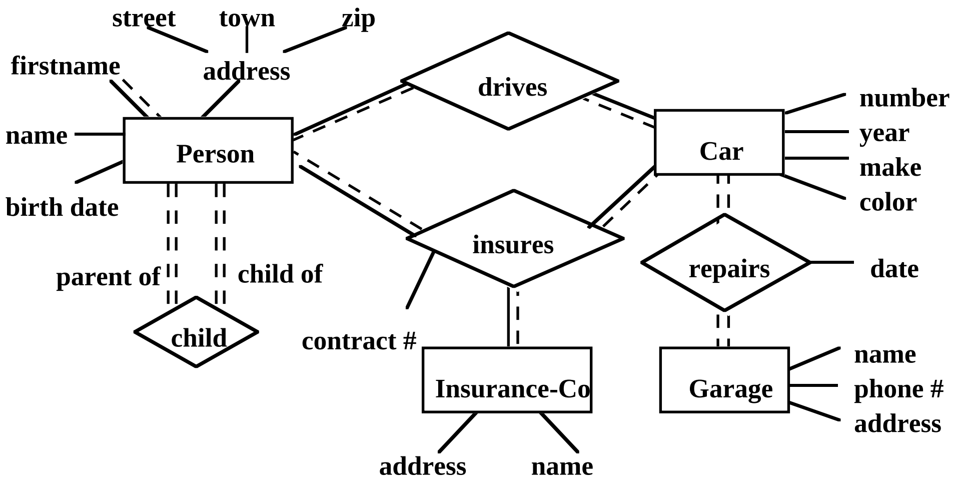


Figure 1: an example of an ERC+ schema

The ERC+ model also includes the definition of two formal manipulation languages for querying an ERC+ database: an algebra [33], and an equivalent calculus [34]. These languages are the underlying framework for visual data manipulations in SUPER.
The functionalities provided by the algebraic operators include as usual: *selection* of objects based on a given predicate, *projection* of objects on a subset of their attributes, *union* of the populations of compatible (approximately, same schema) object types. Specific to ERC+ are the *compression* operator, which allows to discard duplicates in the population of an object type, and the *reduction* operator, which allows the elimination of the values of an attribute which do not satisfy a given predicate.
Most important is the *relationship-join* (*r-join*) operator. Let E1, E2, …, En be the set of object types linked by a relationship type R, the r-join of E1 with E2, …, En via R builds a new object type (and the corresponding population) whose schema includes the schema of E1 plus an additional attribute, named R, whose components are the schemata of R, E2, …, En. In some sense, this operator groups into a single object the information scattered over objects linked by a relationship. A *i-join* operator allows joining object types participating in a given multi-instantiation link [35].
The *product* operator is similar to a relationship join, but associates to each occurrence of the first object type all occurrences of the second object type. Finally, two operators, *renaming* and *simplification*, are used for syntactic manipulations.
Every operation results in the creation of a new object type, with its attributes, environment and population derived from the operands through specific rules. Operations may thus be combined into expressions of arbitrary complexity.

4. SUPER Visual Query Editor

SUPER is intended to present users with a consistent set of visual interfaces which allow definition and manipulation of ERC+ databases. User's specifications are automatically translated onto the underlying DBMS (an OO DBMS in the current prototype), which makes DBMS peculiarities transparent to users. SUPER insists, in particular, on features like *flexibility*, *reusability* and *backtracking*. Flexibility allows users to follow their own strategy. Whenever actions are not atomic, users are allowed to start an action and move to another one without being required to complete the first one. For example, during the schema design process, users are allowed to leave definitions incomplete. During the query formulation process, the user is allowed to independently specify the different components of a query and modify them at any time. As for reusability, users are allowed to reuse definitions of queries already known to the system. This is useful both in schema design and data manipulation. Users may need to reuse existing definitions either in the current schema or in another one. An already evaluated query may be reused to build a similar one. Finally, the user may at any time undo erroneous actions and restore the previous state.

The SUPER query editor allows the visual specification of both queries and updates. It is designed for users with an understanding of semantic data structures, but no specific expertise in query languages is assumed. Its purpose is indeed to allow users to easily write simple and complex queries without having to learn the syntax of an object manipulation language. In this section we present the main features of the query editor. Data updates will not be discussed. Restricted to queries, the expressive power of the editor is equivalent to the ERC+ algebra [6]: it supports all queries except for transitive closure and restructuring (nest, unnest, …) operations.

The construction of a query encompasses the definition of:

- which elements of the schema are relevant for the given query (we call this the definition of the query frame). This might include a modification of the actual structuring of these elements, both to suit the query needs and to make sure that the specification can be unambiguously interpreted;
- conditions on the population of the result. In this step the user defines the predicates to be applied to database occurrences, so that only relevant data is selected;
- format of output. This step specifies which data items (attributes) are to be included in the structure of the result.

These three activities are inherent to the query definition process. They are identifiable in any query language. For example, in SQL: the FROM statement defines most of the frame, the SELECT statement defines the structure of the result and the WHERE statement defines the predicates. As the relational model is semantically poor, the WHERE statement also includes a part of the query frame definition, as equality of keys is used to show which links are to be followed.

The complete query design process is rather complex, especially if the user tries to define these three steps in a single task. So we advocate that user friendliness is best achieved by interacting with only one step at a time. Clear separation between the steps alleviates users' mental load and improves the chances of a correct formulation. The sequence of steps is logically meaningful: for instance, predicates cannot be defined before the query frame is determined. However, users should be allowed to go back to previous steps to correct or refine the current formulation. SUPER implements step separation, by using specific windows for the different steps, as presented in the following subsections. 4.1 to 4.3 discuss issues in the definition of a visual query language. 4.4 uses a specific example to illustrate the overall query formulation process. A complete description and discussion of the visual language can be found in [36].

4.1. Defining the frame of the query

The frame of a query is defined by two types of operations. First, the user has to select the types used in the query, then modify some links and object types to create the hierarchical structure that exactly describes the frame of the query.

4.1.1. Getting object and relationship types from the database schema

This is the initial step for all visual query languages (it corresponds to an "open subschema …" command in textual languages). It reduces the schema to contain only those elements that are involved in the query. This process is simply performed through a sequence of "point and click" specifications.

In SUPER, when users first specify the schema that is to be used, the query editor displays the corresponding diagram in a read-only window, called the database schema (DBS) window. The user can choose between two modes. The most common mode is a traditional Copy-Paste that transfers previously selected elements into the second window, called the working schema (WS) window. In a paste operation new elements are not related to elements that were already in the WS. In the second mode, called Expand, users select an element type in the WS window, and then click on an element in the DBS diagram. The click copies the designated element into the WS window, connects it with the selected object type and moves the selection to the element. If no object type was selected in the WS window, the designated DBS element are copied into WS but not attached to any other object type. If a user clicks on a role, the complete relationship type is transferred into the WS. It is possible to click on a distant element if there is a smallest path from the selected object type to this element. For instance, if the user clicks on a relationship type which has two roles leading to the object type selected in WS, the tool cannot determine which role is to be attached to the selected object type, so it does not transfer anything at all. In this case the user has to click on the role.
Some automatic selection is embedded in SUPER: Object types and relationship types are copied with their attributes; a click on a relationship type will also copy the participating object types and roles.
The DBS window is kept open and unchanged, so that users may later pick additional elements, as needed.

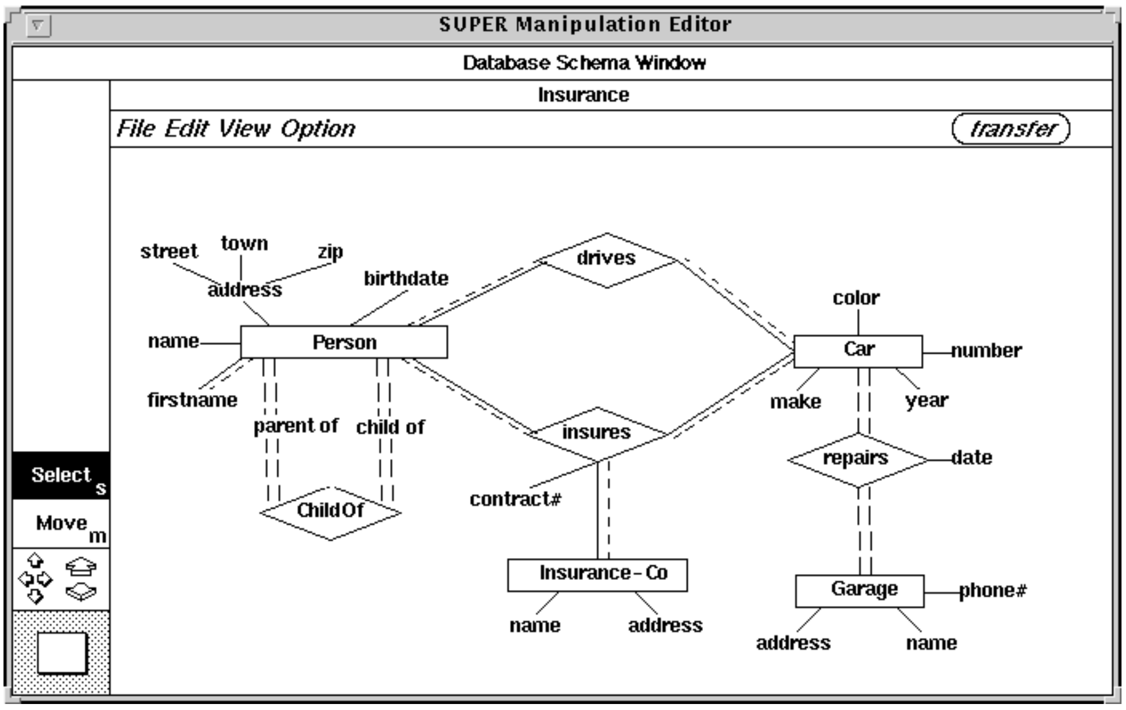


Figure 2: the database schema window

4.1.2. Shaping the frame

The data model on which SUPER is based supports complex objects. The result of a query is a population of objects defined by a complex (nested) type and satisfying the predicates associated to this type. This is different from tools where the resulting population is formed by a flat join of the selected occurrences from all the object types in the query frame.
Elements copied or transferred into the WS window describe a graph which is not necessarily fully connected nor acyclic. As the result of the process of writing a query is an algebraic expression, which cannot include recursion nor transitive closure, the frame of the query is a tree. Note that this still allows expressing cyclic queries. For instance, the query "select drivers who insure the cars they drive" calls for a frame built as the linear sequence of schema elements *Person-drives-Car-insures-Person* and a predicate stating that the person in the first *Person* class must be the same as the person in the second *Person* class. To define the result as a single object type, the WS graph has to be transformed into a tree. To achieve this transformation three types of operation are required, removal of cycles, connection of disconnected subgraphs and selection of a root for the tree.

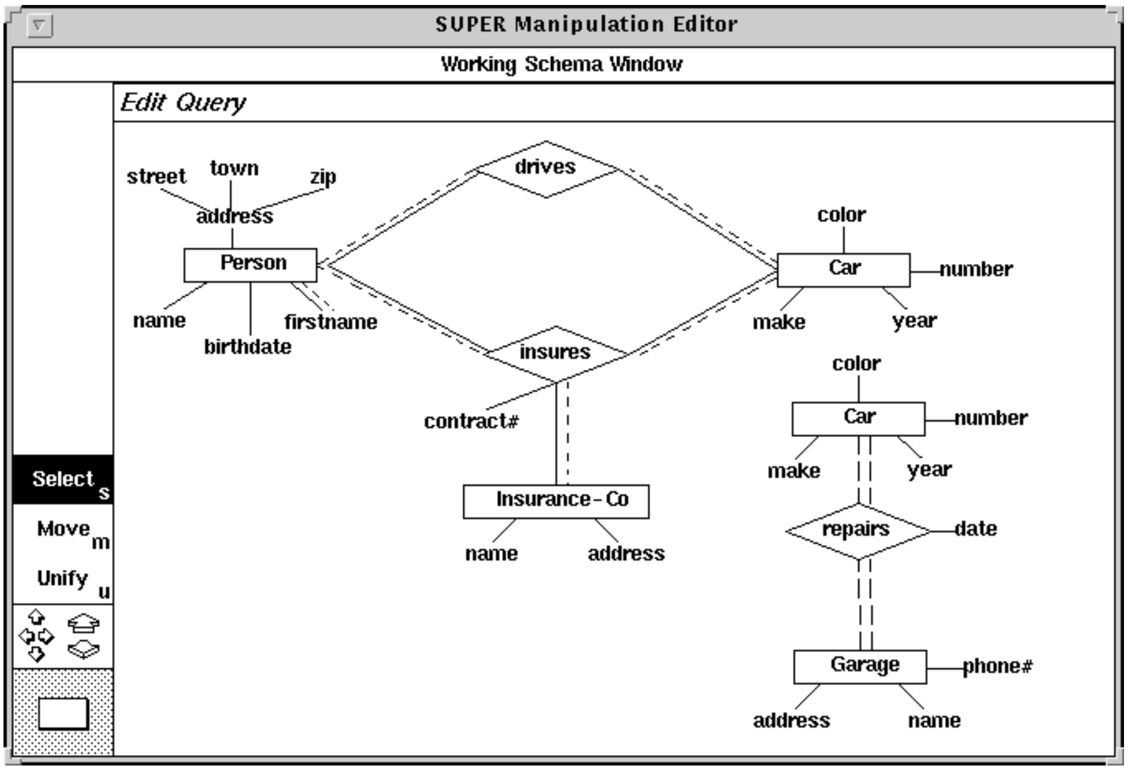


Figure 3: a working schema window showing a cyclic, non-connected graph

First, cycles must be removed. This cannot be an automatic process. There are as many possible interpretations as there are possible duplications of object types involved in the cycle. Such an ambiguity is not acceptable. As the editor cannot infer which interpretation is the intended one, users must explicitly direct the transformation to be performed. In the SUPER query editor, the user can break cycles by removing some links or some nodes of the graph or by disconnecting some links.

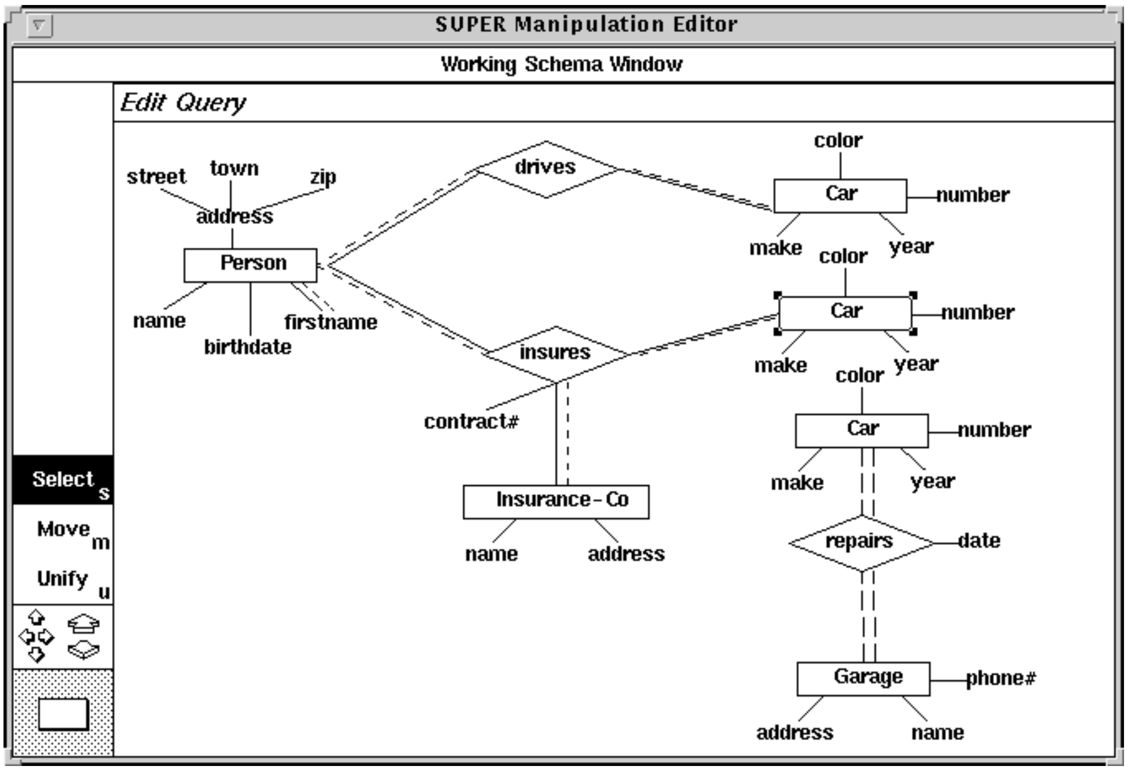


Figure 4: the WS (of fig.3) after disconnection of *insures-Car* role

Disconnection means that the link is detached from the linked object type and attached to a (new automatically created) copy of that object type. While disconnection provides a way to break connections, conversely unification of two object types permits to merge two different copies of the same object type into a single copy with all the links and attributes of both copies. This operation allows the user to connect two disconnected sub-graphs. Figure 4 represent the state of the working schema after the disconnection of *'insures-Car'* role, this is achieved by direct manipulation of the role (shift double click on it). The unification operation is somewhat the opposite operation of disconnection, it is obtained in dragging a representation of an object type over another representation of the same object type. For instance, the *Car* object type related to *Garage* through *repairs* is dragged over *Car* related to *Person* through *insures*.

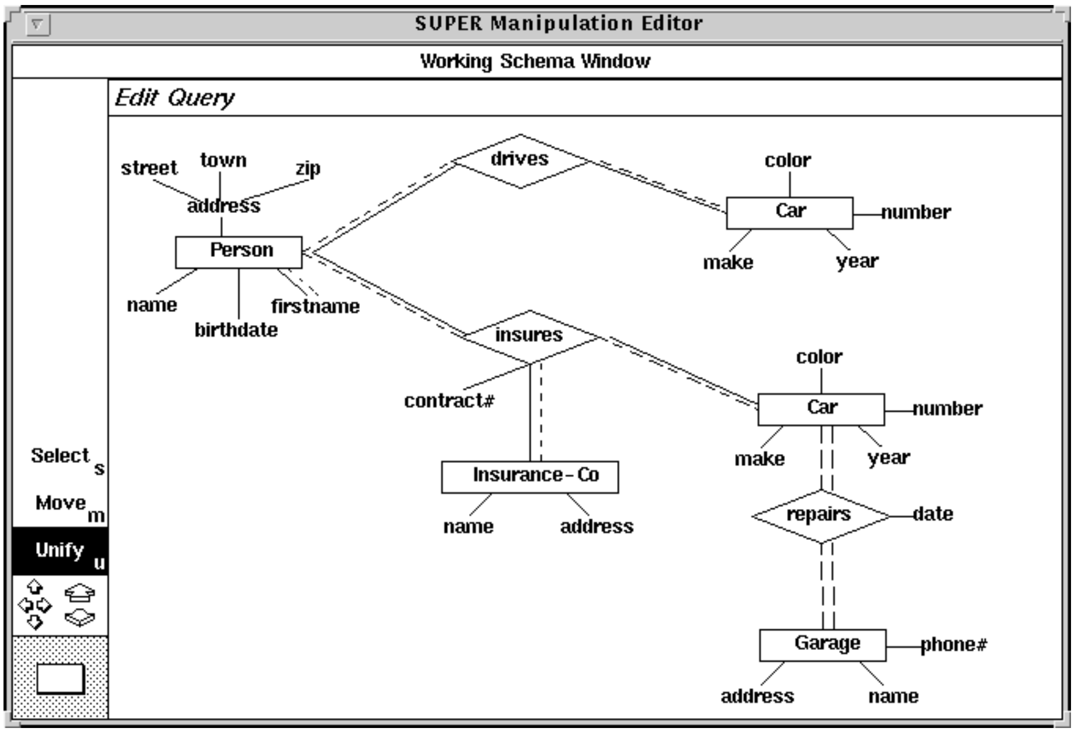


Figure 5: the WS after unification of two copies of the *Car* object type

The disconnection and unification operations allow to transform any graph in an acyclic connected graph. We need an additional operation to transform the new graph into a tree. This operation is done by designating (double click) one of the object types in the WS as the root of the tree. The transformation is automatically done by the tool.
These three operations provide most of the functionalities needed for the proper definition of hierarchical queries over an ER-like schema. Another facility, pruning, is used to remove schema elements (attributes, object types, …) which are not used in the query (neither appearing in the result nor appearing in a predicate). There are some additional facilities to create new elements in the WS: for instance, an artificial object type can be added as a supertype of two existing object types, to represent the union of the two object types. In the same way, it is possible to create an artificial relationship type between two object types, which is defined as the connection of all occurrences of one type to all occurrences of the other type: this relationship type represents a product. These artificial definitions are a very convenient way to specify new classification schemes, which may thus be dynamically introduced when they have not been included in the initial database schema.

4.2. Formatting the output

Once the user has created a correct query frame in the working schema, SUPER builds the corresponding hierarchy as a single object type, with all other informations as attributes. This resulting structure is displayed in a third window, called the selection window (SW, see figure 10). The user can visually check if his/her specifications are going to produce the expected result. If this is not the case, the user can immediately return into the WS window to make the appropriate modifications to the query frame.

The selection window contains all attributes of all object and relationship types participating in the query. Not necessarily all of these attributes belong to the query result. The attributes designated by the user as not belonging to the result are turned shaded. They are not physically removed in order to allow the user to use them in the predicate specification or to reintroduce them in the result if the user has a change of mind.

4.3. Specifying predicates

Predicates on complex objects may be rather clumsy. For the simplest ones (comparison of a monovalued attribute with a constant) a visual counterpart may easily be defined, but the representation of complex predicates is often unreadable. Some tools try to provide a visual representation of logical conditions. It has been shown [30] that these representations are difficult to read and ambiguous. Moreover, there are cases where they have no proven equivalence to first order logic.

In short, a predicate is a logical expression related to an element in the query frame, which contains variables defined on other elements in the frame, those variables being bound before they can be used in a comparison. The domain of a predicate is the set of quantified variables.

The element (object type or attribute) related to the predicate has to be present in the query frame. If this element is the root object type, the predicate defines a selection to be performed on occurrences of the root. If the predicate is attached to an attribute, it defines which values have to be eliminated from the set of values of the attribute (i.e., a reduction operation in terms of the ERC+ algebra). There is a logical dependency between variable definitions and frame. More precisely, if the data model is in first normal form, the set of variables and the frame can be seen as the same object. We illustrate this on an example. Let us assume we want to write the query: persons having children born in 85 <u>and</u> in 90.

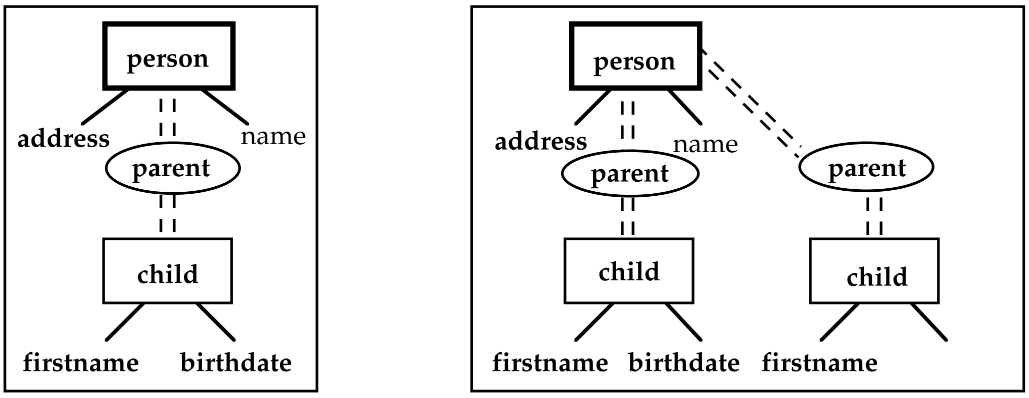


Figure 6: the database schema (left) and the query frame (right) for a first normal form ER model

Figures 6 and 7 show two database schemata, corresponding to ER models, the first one with first normal form attributes, the second one with complex and multivalued attributes. As shown in these same figures, the corresponding frames for the same query are very different.

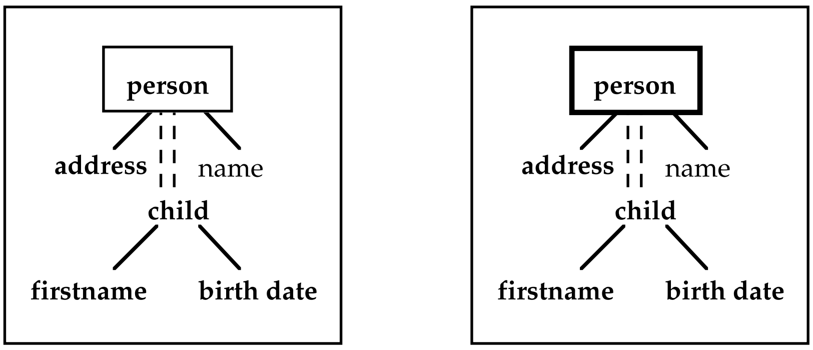


Figure 7: the database schema (left) and the query frame (right) for an E/R model with complex and multivalued attributes

In the first case (Fig. 6), the user can use the same object type twice (*child*) to represent the use of a second variable on this type. In the second case (Fig. 7), the frame cannot include two representations of the attribute *child*. The definition of a second variable is done in the query structure through duplication of the *child* attribute. Generally speaking, every attribute must be represented in the selection window as many times as it appears in the predicates. In particular, attributes which appear only once in the predicates can be directly used as variables. Defining variables on the structure seems very convenient, as it permits the user to write simple queries without thinking of the differences between variables and attributes. Figure 8 shows the content of the selection window with the appropriate variables.

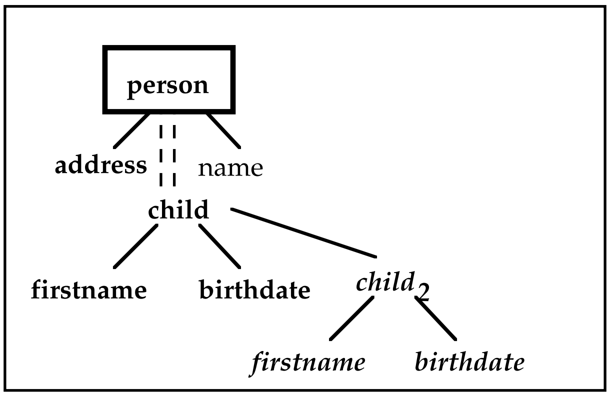


Figure 8: structure of the query with the two variables defined on attribute *child*

4.4. An example query

This section illustrates the process of query formulation in SUPER. Let us assume that the user wants to formulate a query against the schema shown in figure 2. The corresponding diagram will be displayed in the database window. If the intended query is:

Select name and address of people who insure a 1984 Ford,

the user will pick the relationship type *insures*. It will be copied into the WS window, together with the linked object types (*Person*, *Insurance-Co* and *Car*) and all their attributes (Figure 9).

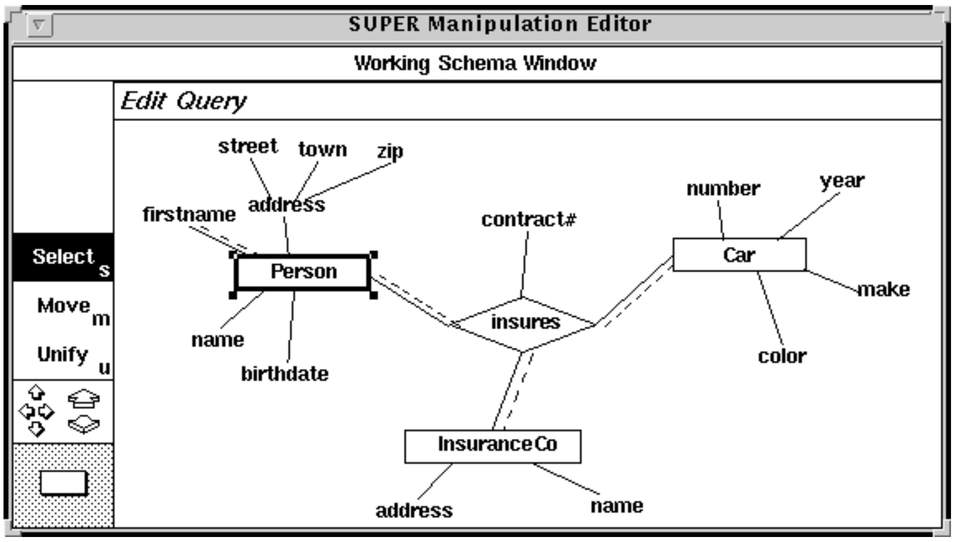


Figure 9: query editor window for the query frame definition process

Next, assume the user designates *Person* as the root of the hierarchy. The resulting selection window is shown in figure 10.

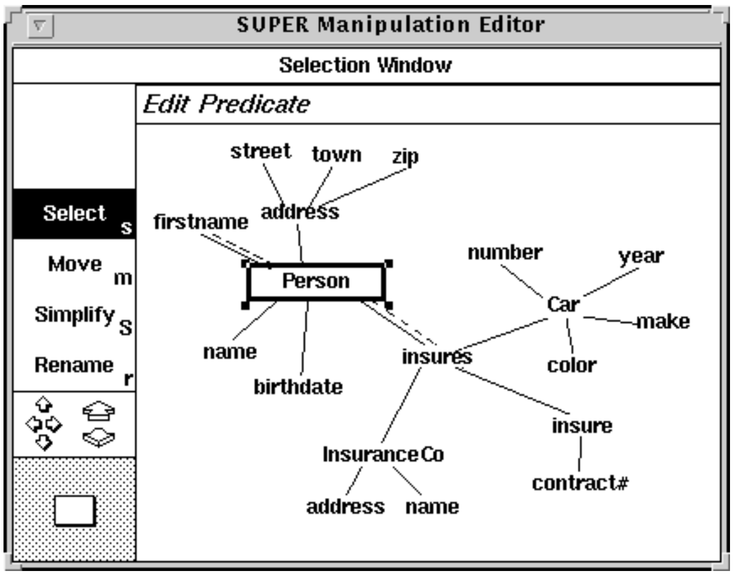
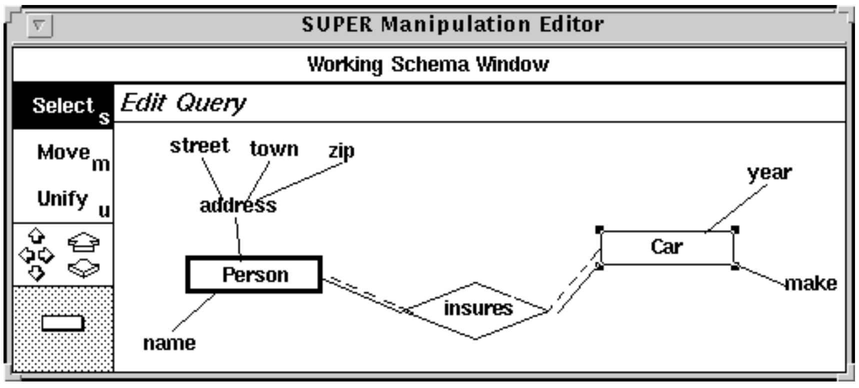


Figure 10: the resulting hierarchical structure, showing the root as a box

The user easily recognizes that this structure includes many attributes (s)he is not interested in. Consequently, (s)he will return to the WS window and prune useless elements. The attributes of interest to this query are the *name* and the *address* in *Person* (to form the result) and *make* and *year* in *Car* (for predicate specification). Pruning will result in new displays, as shown in figure 11 below.



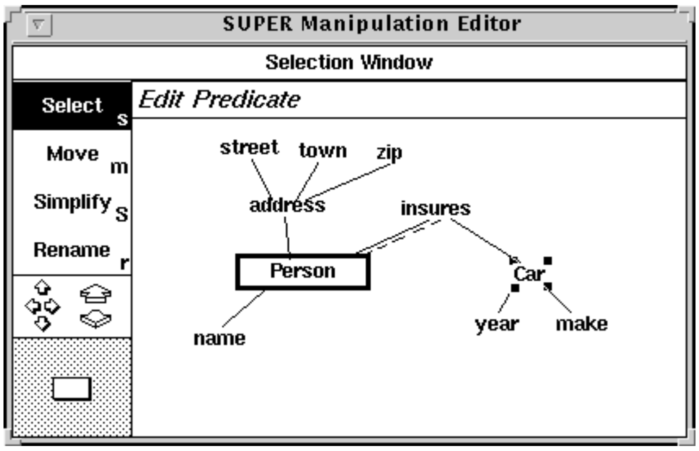
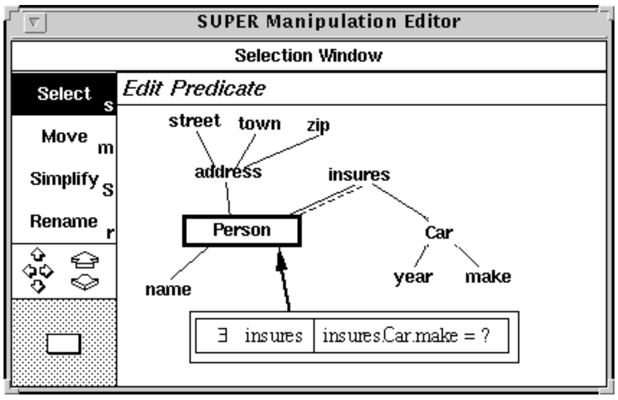


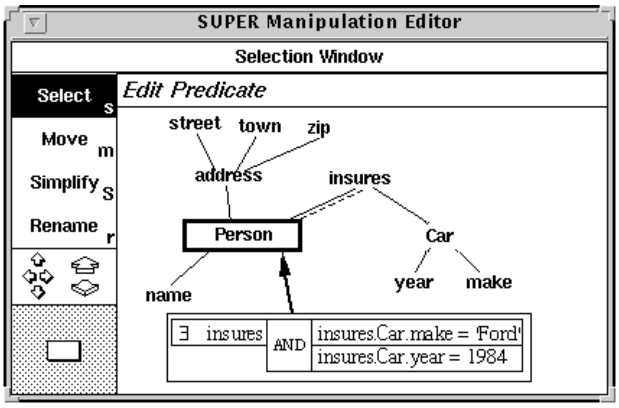
Figure 11: the updated query frame and the corresponding hierarchical structure

The user will now tell the editor that (s)he wants to define a predicate. A predicate box is displayed in the selection window. As a default policy, the predicate is associated to the root. The user designates the attributes (*make* and *year*) involved in the predicate. The 'exist *insures*' clause is automatically generated (but modifiable) by the editor, because of the multi-valuation of the *insures* attribute, to which *make* belongs (Figure 12).



∃ insures	insures.Car.make = ?

Figure 12: SW after designation of *make*



∃ insures	AND	insures.Car.make = 'Ford'
		insures.Car.year = 1984

Figure 13: the final state of the SW

While designating the second attribute (*year*), the user also has to specify the logical connector between the two predicates. The predicate box now contains all the necessary quantified attributes. The user has to complete the specification by entering the appropriate values: "Ford" for *make* and 1984 for *year*. The predicate is now displayed as in figure 13.
Let us now suppose that the user wants to proceed with the specification of another query, very similar to the previous one:

name and address of persons who insure a 1984 Ford, and who also insure a car that has been repaired in the "Morris" garage.

The user will return to the working schema window and import, from the DBS window, the additional needed information (the relationship type *Repairs*). The result is shown in figure 14.
The next actions will be to unify the two *Car* object types and to prune useless elements (figure 15). The first predicate can then be reused, to select 1984 insured Ford (figure 16).

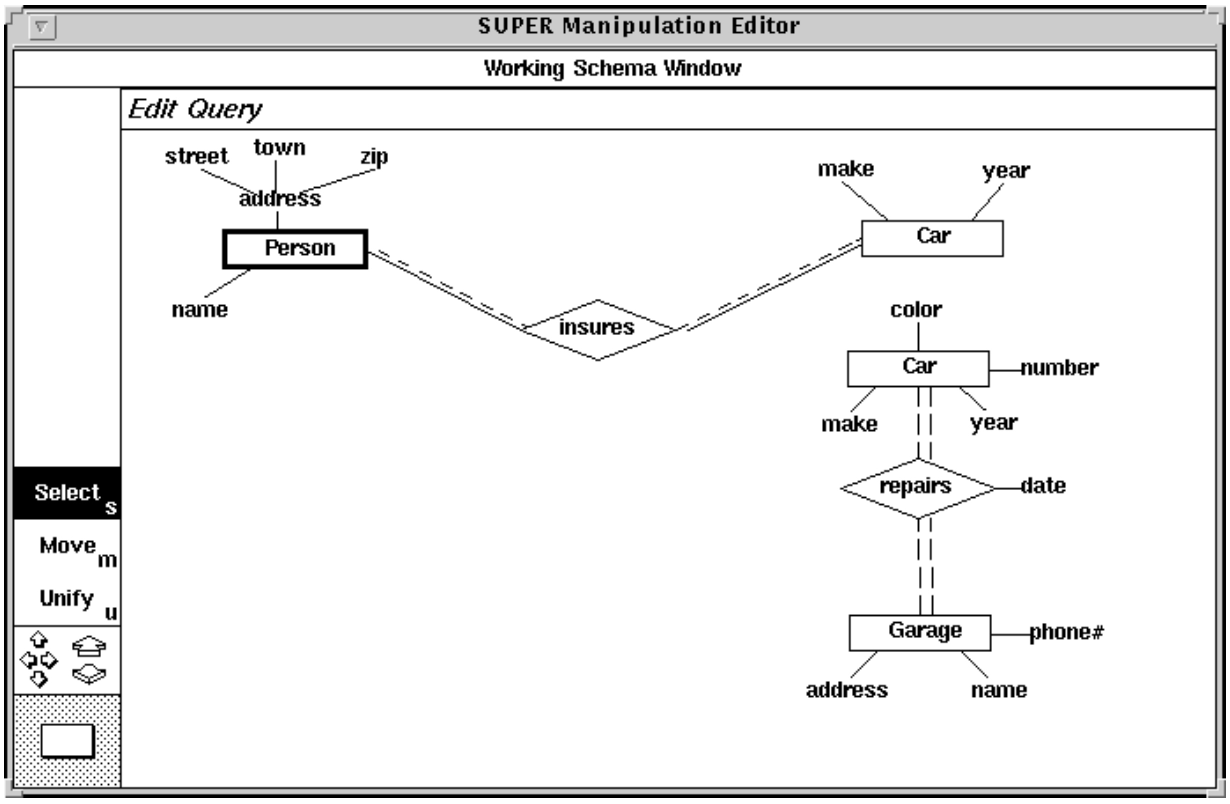


Figure 14: starting a new query through modification of the current query

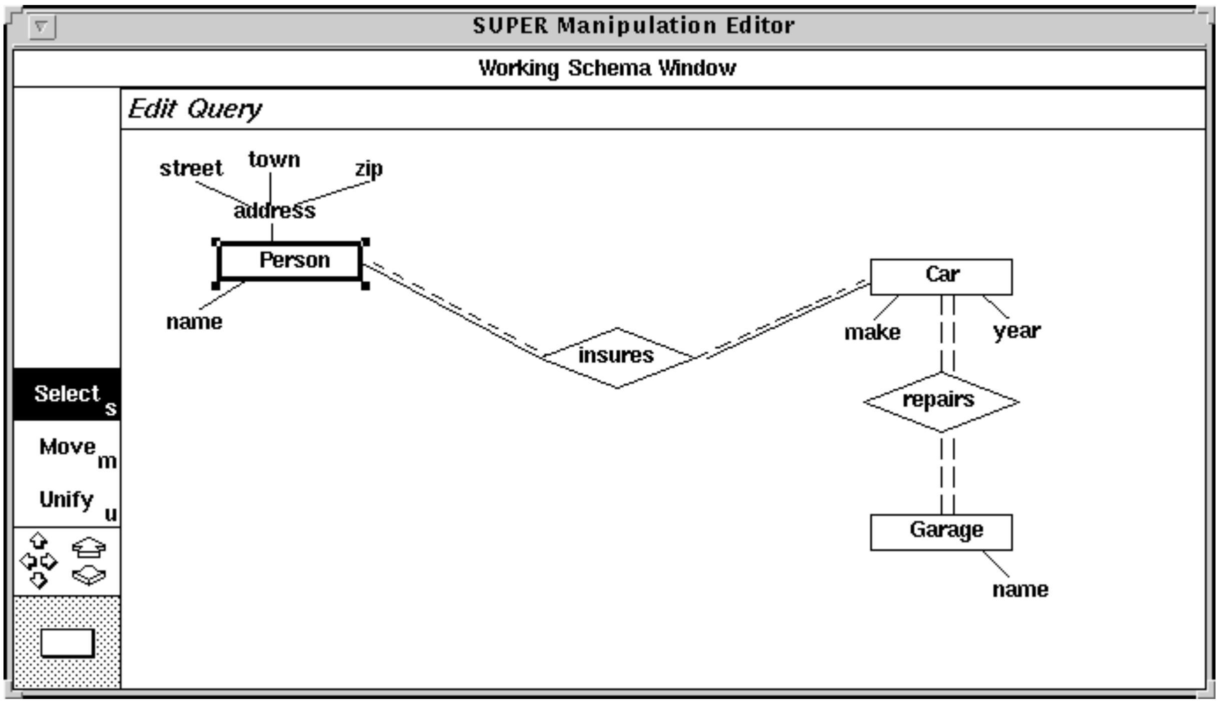
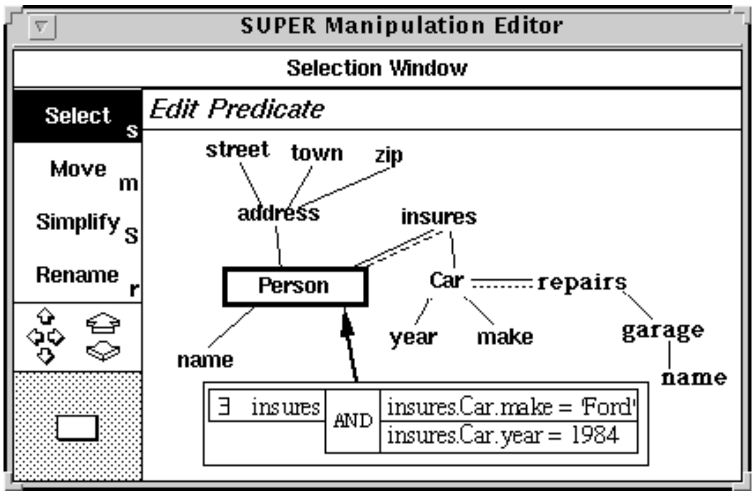


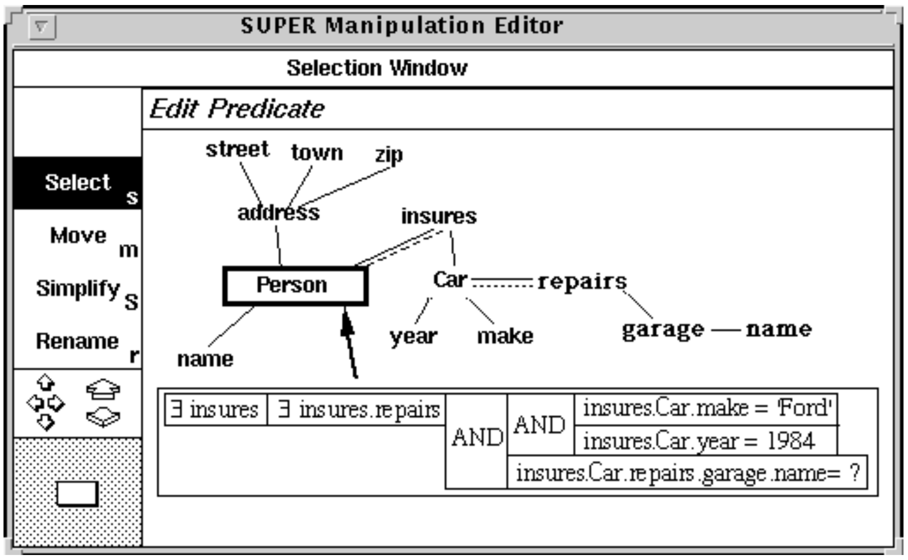
Figure 15: keeping useful objects



∃ insures	AND	insures.Car.make = 'Ford'
		insures.Car.year = 1984

Figure 16: the previous predicate is kept unchanged on the new structure

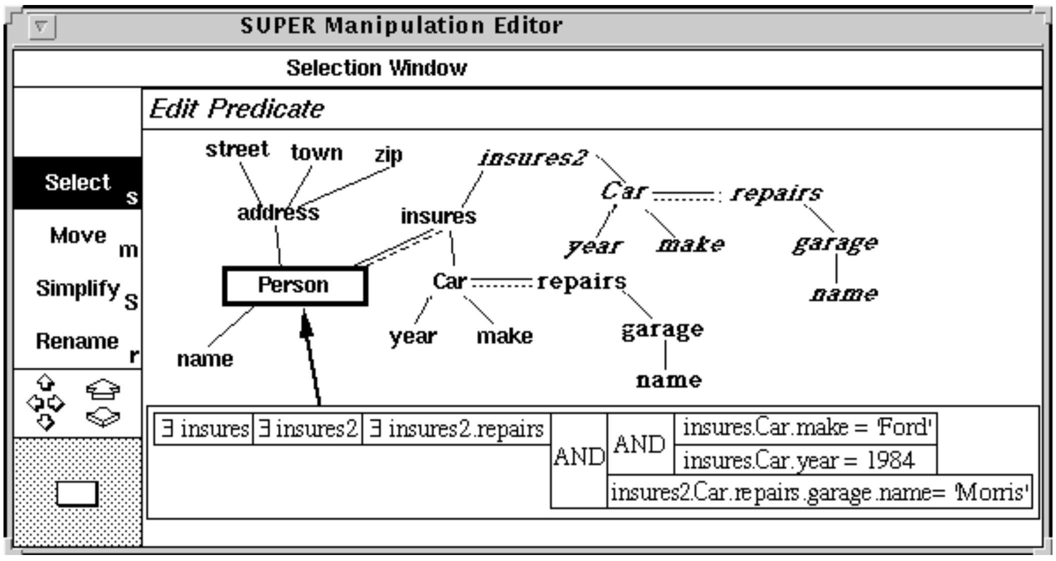
The user has to refine the predicate. To fulfill this operation (s)he has to introduce a variable (the garage name) in the predicate box. If the user drags the attribute *garage* into the predicate box, (s)he introduces a variable defined on this attribute and the system produces the following predicate:



∃ insures | ∃ insures.repairs | AND | AND | insures.Car.make = 'Ford'
insures.Car.year = 1984
insures.Car.repairs.garage.name= ?

Figure 17: the predicate as automatically modified if *garage* is dragged into the predicate box

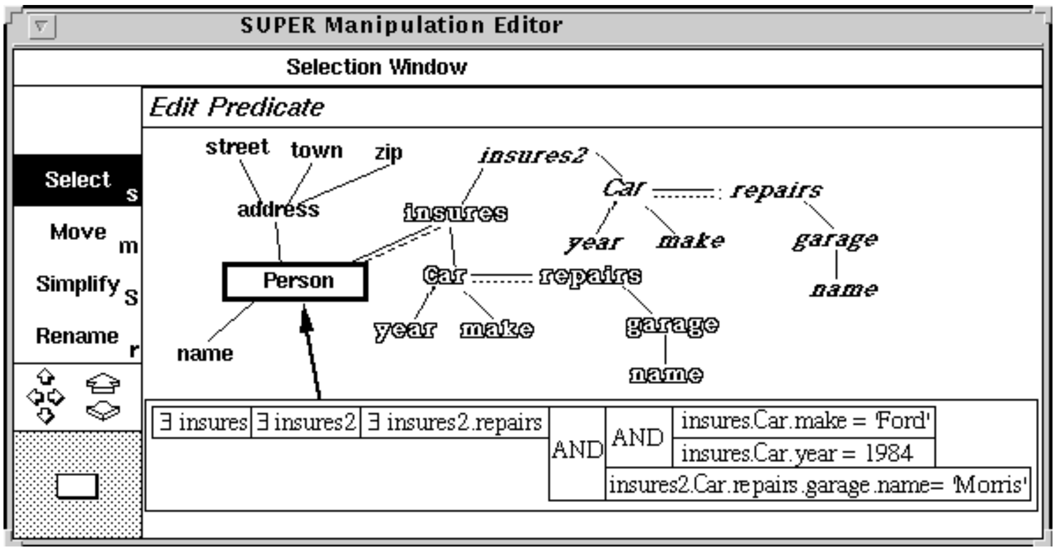
This is not really what is wanted: the predicate selects persons having a 1984 Ford car repaired at some garage. The user can correct this by removing this new condition, which leads back to the state shown in figure 16, and by defining a new variable on the attribute *insures* to denote that the Ford car and the repaired car do not have to be the same car.
The introduction of the 'good' variable insure2.Car.repairs.garage.name in the predicate box (and of the 'Morris' value to replace the question mark) leads to the following state shown in figure 18.



∃ insures | ∃ insures2 | ∃ insures2.repairs | AND | AND | insures.Car.make = 'Ford'
insures.Car.year = 1984
insures2.Car.repairs.garage.name= 'Morris'

Figure 18: query with the desired predicate

The second condition (insuring a car that has been repaired in the "Morris" garage) involves the *insures* attribute independently from the first condition (*insures* is multivalued). Its specification calls for the duplication of *insures*. The predicate can now be expanded to include the new condition (figure 18). Notice that *insures-2* and its *repairs* components have been existentially quantified by the editor according to the default rule.
The query is almost finished, the user only has to remove undesired attributes from the structure. A simple hiding click on the *insures* attribute leads to the final representation of the query (figure 19).



∃ insures	∃ insures2	∃ insures2.repairs	AND	AND	insures.Car.make = 'Ford'
					insures.Car.year = 1984
				insures2.Car.repairs.garage.name= 'Morris'	

Figure 19: the new query ready for evaluation

Additional functionalities supported by SUPER allow the storing of queries for later reuse or modification, as well as evaluation of partial queries. The latter is useful for query debugging. Suppose the user is confronted with a result different from what (s)he expected, without recognizing the problem within the formulation. Then the query can be broken down into two or more separate queries by disconnecting some links. Independent evaluation of the sub-queries can be performed to identify the changes that have to be made. After this refinement, the original corrected query can easily be rebuilt by unification of the duplicated object types resulting from the previous disconnection.
Once the query is defined it can be inserted into the database schema window where it appears as an object type related to its root object type by a derivation link. It can be reused in a query or browsed as any object type, it inherits all links of its root object type.

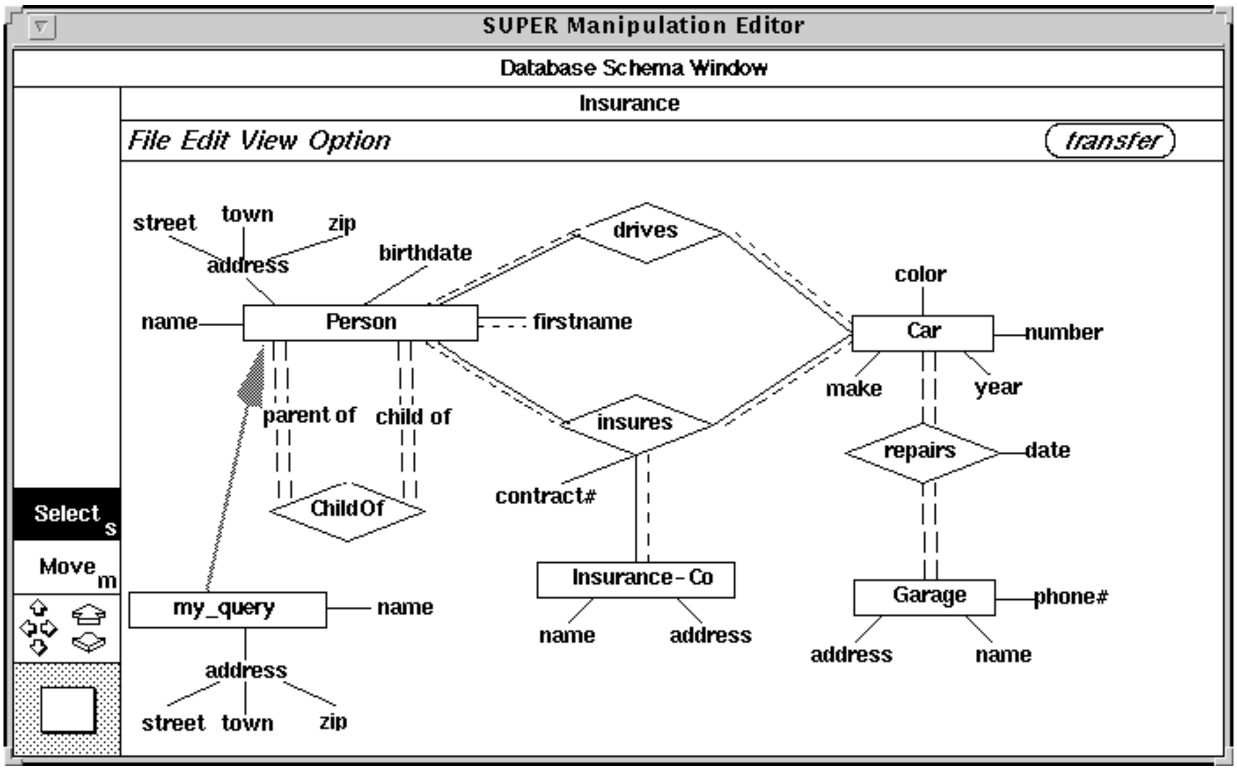


Figure 20: the database schema with the query

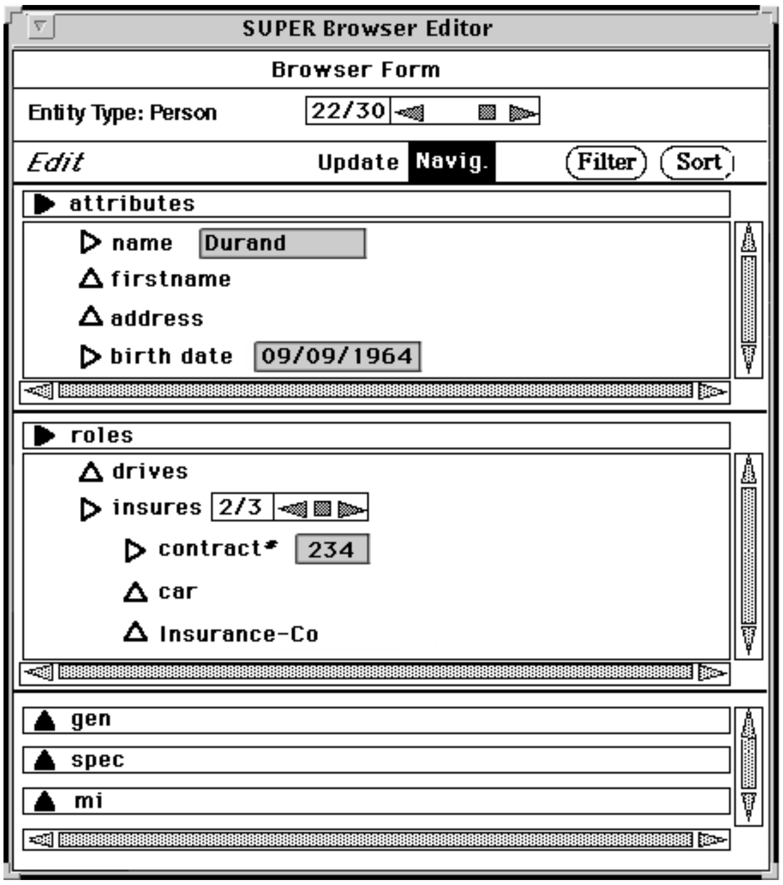
5. THE SUPER BROWSER

The SUPER browser provides two display modes for data visualization. In the **forms-based** mode, occurrences are presented with forms-like representations of the corresponding object or relationship type, while in the **graphical** mode they are presented with entity-relationship-like diagrams showing the occurrences currently being examined. These occurrences can be directly manipulated by the user.
Regardless of whether forms or diagrams are used, each browsing session has an associated state that indicates if the session is in the **navigational** mode or in the **update** mode. In navigational mode, it is possible to browse the occurrences of a population (one at a time), to examine the values of their attributes and to move from an occurrence to another one belonging to a related population (related via a role or a multi-instantiation link). In the update mode, occurrences can be modified, deleted and created. Anytime during a session the user can switch between navigational and update mode with a click on a button. For instance (s)he can start browsing the database in the navigational mode, look for some data to be updated and switch to the update mode when the desired data is found.
A browser session starts with the selection of the database to browse and the choice of the visualization mode. As we said in the previous session the user can browse the result of a query. In this case the browser session starts from the database schema window on the database schema augmented with the query (e.g. figure 20).
Then, the user has to specify the entry point to the database, i.e. the type whose population the user wants to browse first. Any type can be selected as entry point. A window named *browsing window* is displayed, containing some general commands concerning the browser session. Depending on the visualization mode, this window has a different look and a somewhat different semantics. In the graphical mode it consists of a graphical palette, a pull-down menu and a workspace where the occurrences are displayed. In the forms-based mode, only the pull-down menu is available since most of the buttons to realize the operations are made available in the forms.
The population of the entry point becomes the active population and its first occurrence becomes the active occurrence. With active population we mean the set of occurrences of an object type or a relationship type that users can browse. In general it is a subset of the entire population. Users can browse the active population, enlarge or reduce it and navigate towards other populations. Furthermore we consider as active occurrence the instance of the active population that is to be inspected.
The basic principle of our data browser is the following: users start browsing on the population of an entry point, and from there they can examine the related instances of the database by following the roles and multi-instantiation links. For instance, on figure 1, users can start browsing the object type *Person*, look for a particular person, navigate through the relationships *insures* towards the *Car* this person insures. If he/she insures more than one car, since the active population of the relationship type *insures* is the set of instances linked to the starting one, users can browse on them.
It is possible to come back to a previously inspected population to continue the navigation through a different path. Since during the navigation the cycles are broken, the set of inspected types can be visualized as a tree structure whose entry point is the root. At any time users can designate a node in the tree as the new entry point in the database. The population of the designated type becomes the active population, the sub-tree having the new entry point as root is maintained while the other nodes are released.
The data browser also provides additional facilities, namely: sorting, filtering, synchronization and editing. The sort function is provided to enhance the readability of the active population. It allows to sort the instances of an attribute in ascending or descending order. For example, before a set of persons is browsed, users can sort the persons in alphabetical order, or in ascending order on the birthdate attribute. In a similar way the set of values of multi-valued attributes can be sorted.
Filtering allows users to select which occurrences in a population are to be examined by taking advantage of the visual query editor. While browsing an object type , users can call the query editor to perform a select on the active population. The structure of the query is implicitly defined, thus the first step of the query editor is skipped; the type becomes the root of the hierarchy and appears in the selection window. The query formulation process is then performed as for any other query. Finally the query is evaluated on the active population and the result returned to the data browser.
As already mentioned, through navigation users can reach all the instances connected with the starting population.
With synchronization we mean that, once users have produced the tree of types, any operation applied on a type of the tree is immediately and automatically propagated through the tree.
Editing allows users to modify, delete and create instances and to establish a connection between occurrences of two object types through a multi-instantiation link, or through relationship type

occurrences. To edit instances, users have to switch to the update mode; field values become editable and create and delete commands become active. An easy way to define a relationship between occurrences belonging to different populations is to browse independently the object type populations, position on the appropriate occurrences and finally indicate the relationship type which will link the occurrences. For instance, on the schema shown in figure 1 to define a new insurance, users browse the populations of *Person*, *Car* and *Insurance-Co*, choose occurrences and create an instance of the relationship *Insures* which links the previously selected objects. The system will then ask for the insurance contract number. Connections using multi-instantiation links are defined at creation time. At any time it is possible to specialize an occurrence.

5.1 Browsing through forms

In the *forms-based* mode the active occurrence is displayed in an object form. It consists of the command area, the attribute area, the role area and, if the entry point is an object type, the multi-instantiation area.


SUPER Browser Editor

Browser Form

Entity Type: Person 22/30

Edit Update **Navig.** Filter Sort

attributes
- name: Durand
- firstname
- address
- birth date: 09/09/1964

roles
- drives
- insures 2/3
  - contract*: 234
  - car
  - Insurance-Co

gen

spec

mi

Figure 21: the form browser

The command area includes a horizontal scroller to browse the set of occurrences, the number of occurrences of the active population, the relative number of the active occurrence and the buttons to sort and filter the active population.
In the attribute area the names of the first level attributes are listed and the values of simple mono-valued attributes are displayed; complex and multi-valued attributes are *closed*. Users can *open* them by clicking on the associated button; the structure of the complex attributes is thus displayed and a horizontal scroller is associated with the multi-valued attributes to browse their values.
In the role area the names of the roles of the object type are listed. They are initially *closed* and they can be *opened* in two different ways. With a click on the role users can see the instances linked to the active occurrence without opening a new object form, i.e. without changing the active population. These instances are displayed in the role area. If the user clicks twice on the role, the set of occurrences linked to the active occurrence becomes the active population, its first instance becomes the active occurrence and it is displayed in a new object form.

The multi-instantiation area is defined only for the object type forms and lists the names of the object types linked through a generalization/specialization or multi-instantiation link. As for the roles users can inspect the linked occurrences or navigate to them.

5.2 Graphical browsing

In the *graphical mode* the functions are the same as in the forms-based mode but realized with a direct manipulation approach similar to the ERC+ diagram. The entry point is shown in the browsing window with a graphical symbol derived from the diagrammatic construct of the corresponding type (figure 22).

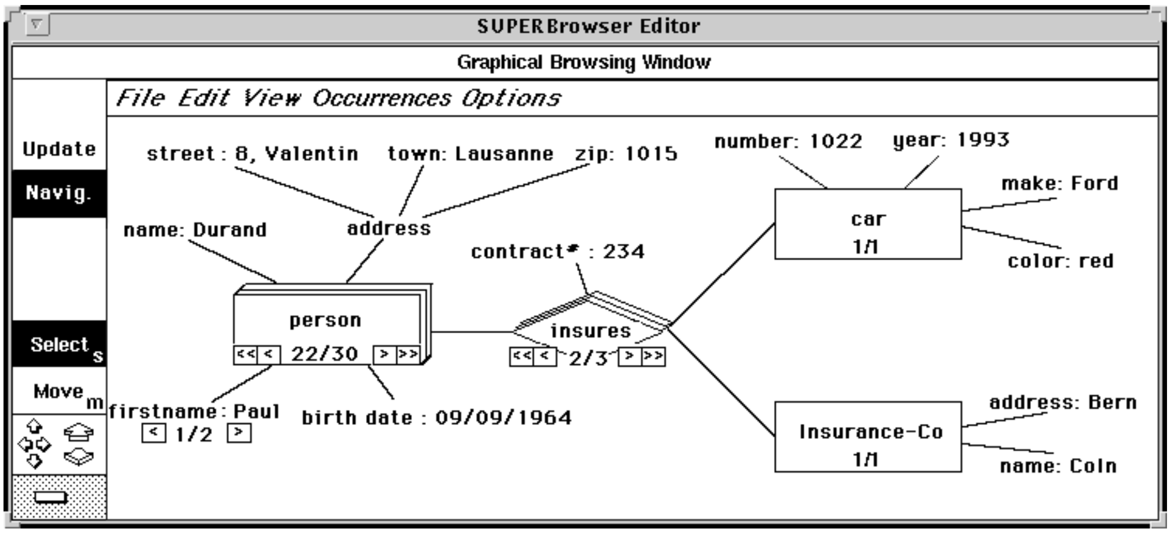


Figure 22: the diagrammatic browser

The entry point is initially represented with attributes and values, four buttons to browse the population and two numbers indicating the number of instances of the active population and the relative position of the active instance.

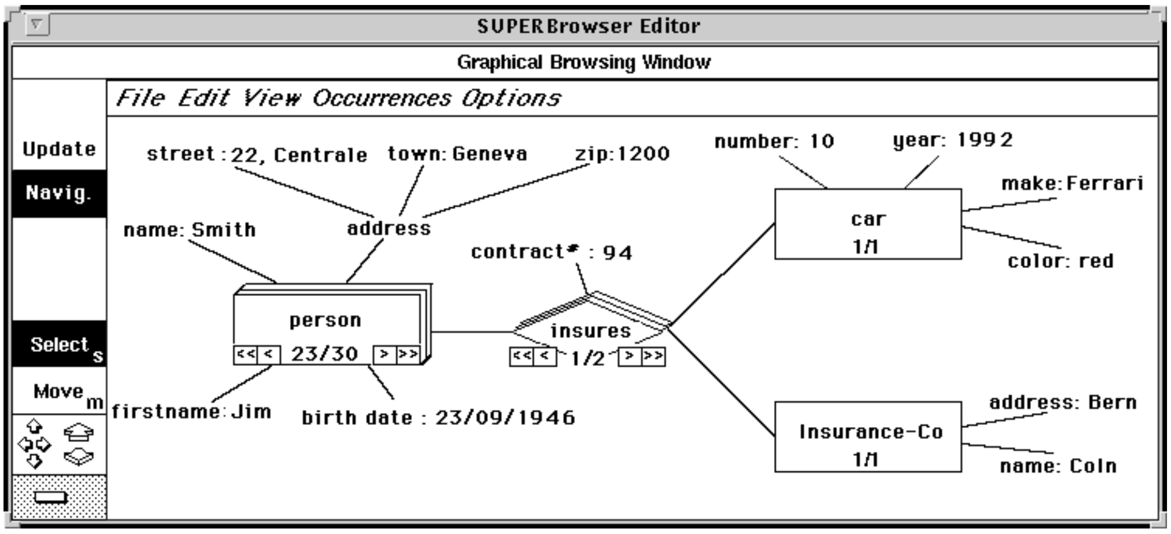


Figure 23: the diagrammatic browser after the get next

For example, to display the next person the user clicks on the button get next '>' of the object type *Person*. The next instance becomes the active instance and according to the synchronization facility, the update is propagated to the *insures* relationship and, from this, to the object type *Car* and *Insurance-Co* (figure 23). The roles and the multi-instantiation links are shown only after users have specified that they want to navigate through the database. These links are mouse sensitive so by pointing at them users indicate the path to follow.

As the browser is used to manipulate data, in the navigational mode the structure of an attribute is displayed only if values exist for the current occurrence. In the same way, roles and multi-instantiation links are displayed only if the active occurrence is linked to other occurrences through these links.

5.3 Conclusions

The SUPER browser turned out to be a powerful, easy to use tool for a quick investigation of database instances. In particular experience has proved that the graphical browser is an accessible tool also for new users, since they tend to relate better to the graphical representation. With the forms-based browser users should not inspect many population at the same time to avoid proliferation of windows.

6. THE PROTOTYPE

The SUPER prototype has been developed in a UNIX environment on SUN workstations, using C++. A graphical toolkit from Stanford University, Interviews (later Unidraw) has been used to facilitate the implementation of the visual interface. ONTOS served as underlying DBMS; a transfer to O2 is being considered, as well as another version to run on top of INGRES.
The first specification of the SUPER editors has been elaborated in cooperation with a group of ergonomists at BULL France. We believe that such a cooperation has been extremely instrumental in leading us towards what we see as really user-oriented specifications. These have been drawn without ever considering implementation aspects. Another major benefit has been that the implementation phase did not call for any reengineering of the specifications. Lack of reengineering has also been possible thanks to the well established background in terms of formal definitions of the ERC+ data model and algebra.
Implementation has been carried in parallel on interface aspects (design of the screens and of the interactions) and on the underlying data manager. The latter is what we call the ERC+ abstract machine: a software layer that is capable of storing an ERC+ schema, and data described by such schemata. The ERC+ abstract machine constitutes a layer that makes the other tools in SUPER independent of the DBMS being used. Expressions in the ERC+ DDL and DML are transformed into a generic format that is either used directly in our interpreter or is translated to a particular SQL dialect by a DBMS-specific module. In the version of the ERC+ abstract machine that uses ONTOS, an algebra interpreter has been implemented in C++. Queries are optimized by avoiding the creation of intermediate results in nested queries, and by query restructuring: In the version using the relational model, we use DDL/DML implementation of the relational DBMS.
The prototype has been demonstrated at various conferences and to visitors to the laboratory. The feedback has been positive, but we believe no significant conclusion can be inferred before a real experiment is possible with actual users in a real application environment. The observation we made is that there seems to be a definite preference for understandability of interactions versus power of functions and richness of displays.

7. CONCLUSIONS AND FUTURE WORK

We have presented the query editor and the browser which are currently included in the SUPER toolset. SUPER aims at providing a complete design support throughout the database life cycle. Important features of the toolset are flexibility, which allows each user to follow his own design strategy, reusability, which allows the user to benefit from previous work, and backtracking, which allows to undo erroneous actions.
We have described a visual query editor that is based on an assertional approach, i.e., a user specifies the different components of a query independently of other components. The specification and execution of a query includes the following actions; selection of the sub-schema relevant for the query, transformation of this sub-schema into a hierarchical structure, specification of predicates, definition of the resulting type, and presentation of the result. The process is all visual, including the predicate specification.
The data browser allows navigation in the database. In scrollable windows, the user can see all elements related to a current element, he can manipulate his view of the multivalued complex attribute

structure, and he can choose a different view of the object by switching the type according to the generalization hierarchy. While browsing the database, the user can change values, insert instances or remove single instances or sets of instances.
Also included in the toolset is the design editor, a visual tool for defining ERC+ schemata. It provides two modes of interaction, an alphanumeric mode and a graphical mode. In the alphanumeric mode, the user fills in form-like representations of the different ERC+ concepts. In the graphical mode, the user builds an ERC+ diagram by direct manipulation.
Current work aims at allowing the user to define integrity constraints by constructing event condition action rules. The rules are triggered on insert, delete, update and select. The condition part of the rule can be defined on complex structures in the database. This complexity is mastered by use of the visual query editor to define the joins necessary to gather the information that is needed in the condition predicate. Another tool currently under specification is the view integration tool which is used to build an integrated schema from a set of user views. This tool will constitute the kernel of a future database design environment. Also under specification are tools for view definition, schema normalization, schema restructuring and schema evolution.

REFERENCES

[1] M. E. Senko, FORAL LP: Design and Implementation, in *Proc. of the 4th Int'l Conf. on Very Large Data Bases*, pp. 225-267, 1978

[2] M. M. Zloof, Query By Example, in *AFIPS Conference Proceedings, National Computer Conference*, vol. 44, pp. 431-438, 1975

[3] B. Shneiderman, Direct Manipulation: A Step Beyond Programming Languages, *Computer*, **16**(8), 1983, 57-69

[4] C. Batini, T. Catarci, M. F. Costabile and S. Levialdi, Visual Query Systems, *Technical Report 04.91*, Università degli Studi di Roma "La Sapienza", Dipartimento di Informatica e Sistemistica, March 1991

[5] J. Larson, J. B. Wallick, An Interface for Novice and Infrequent Database Management System Users, *AFIPS Conference Proceedings, National Computer Conference*, vol. 53, pp. 523-529, 1984

[6] C. Parent and S. Spaccapietra, ERC+: An Object Based Entity-Relationship Approach, in *Conceptual Modelling, Database and CASE*, (P. Loucopoulos and R. Zicari, Eds.), pp. 69-86, John Wiley & Sons, 1992

[7] J. Larson, A Visual Approach to Browsing in a Database Environment, *Computer*, **19**(6), 1986, 62-71

[8] W.-F. Riekert, The ZOO Metasystem: A Direct Manipulation Interface to Object-Oriented Knowledge Bases, in *Proc. of European Conf. on Object-Oriented Programming - ECOOP '87*, pp. 145-153, 1987

[9] I. P. Groette and E. G. Nilsson, SICON, an Iconic Presentation Module for an E-R Database, in *Proc. of the 7th Int'l Conf. on Entity-Relationship Approach*, (C. Batini, Ed.), pp. 137-155, 1988

[10] R. Agrawal, N. H. Gehani and J. Srinivasan, OdeView: The Graphical Interface to Ode, in *Proc. of ACM SIGMOD '90, Int'l Conf. on Management of Data*, (H. Garcia-Molina, H.V. Jagadish, Eds.), pp. 34-43, 1990

[11] J. Durand et al., Data Model and Query Algebra for a Model-Based, Multi-Modal User Interface, in *Interfaces to Database Systems*, (R. Cooper, ed.), pp. 311-337, Springer-Verlag, 1992

[12] B. Gulla, A Browser for a Versioned Entity-Relationship Database, in *Interfaces to Database Systems*, (R. Cooper, Ed.), pp. 136-152, Springer-Verlag, 1992

[13] D. Bryce and R. Hull, SNAP, a Graphics-Based Schema Manager, in *Proc. of the 2nd IEEE Int'l Conf. on Data Engineering*, pp. 151-164, 1986

[14] D.H. Fishman et al., Overview of the Iris DBMS, in *Object-Oriented Concepts, Databases, and Applications*, (W. Kim and F.H. Lochovsky, Eds.), pp. 219-250, ACM Press, Frontier Series, 1989

[15] T. R. Rogers and R. G. G. Cattell, Entity-Relationship Database User Interfaces, in *Proc. of the 6th Int'l Conf. on Entity-Relationship Approach*, (S.T. March, Ed.), pp. 323-335, 1987

[16] M. Kuntz and R. Melchert, Ergonomic Schema Design and Browsing with More Semantics in the Pasta-3 Interface for E-R DBMSs, in *Entity-Relationship Approach to Database Design and Querying*, (F. Lochovsky, Ed.), North-Holland, 1990

[17] F. Staes, L. Tarantino, A. Tiems, A Graphical Query Language for Object-Oriented Databases, in *Proc. of 1991 IEEE Workshop on Visual Languages*, pp. 205-210, 1991

[18] M. Gemis, J. Paredaens, I. Thyssens, A Visual Database Management Interface based on GOOD, in *Interfaces to Database Systems*, (R. Cooper, Ed.), pp. 155-175, Springer-Verlag, 1992

[19] H. Ramos, Design and Implementation of a Graphical SQL with Generic Capabilities, in *Interfaces to Database Systems*, (R. Cooper, Ed.), pp. 74-91, Springer-Verlag, 1992

[20] N. H. McDonald, A MultiMedia Approach to the User Interface, in *Human Factors and Interactive Computer Systems*, (Y. Vassiliou, Ed.), pp. 105-116, Ablex Publishing Corp., 1984

[21] G. Ozsoyoglu and H. Wang, Example-Based Graphical Database Query Languages, *Computer*, **26**(5), May 1993, 25-38

[22] Z. Q. Zhang and A. O. Mendelzon, A Graphical Query Language fort Entity-Relationship Databases, in *Entity-Relationship Approach to Software Engineering*, (Davis et al., Eds.), pp. 441-448, North-Holland, 1983

[23] M. Angelaccio, T. Catarci, G. Santucci, Query by Diagram*: A Full Visual Query System, in *Journal of Visual Languages and Computing*, **1**, 1990, 255-273

[24] R. A. Elmasri and J. A. Larson, A Graphical Query Facility for ER Databases, in *Entity-Relationship Approach - The Use of ER Concept in Knowledge Representation*, (P. P. Chen, Ed.), pp. 236-245, North-Holland, 1985

[25] B. Czejdo, R. Elmasri, D. W. Embley, M. Rusinkiewicz, A Graphical Data Manipulation Language for an Extended Entity-Relationship Model, *Computer*, **23**(3), 1990, 26-36

[26] K. J. Goldman, S. A. Goldman, P. C. Kanellakis, S. B. Zdonik, ISIS, Interface for a Semantic Information System, in *Proc. of ACM SIGMOD '85, Int'l Conf. on Management of Data*, pp. 328-342, 1985

[27] H. K. T. Wong and I. Kuo, GUIDE: Graphic User Interface for Database Exploration, in *Proc. of the 8th Int'l Conf. on Very Large Data Bases*, pp. 22-32, 1982

[28] A.T. Berztiss, Query Formulation based on Visual Models of Data Bases, in *Human Factors in Information Systems Analysis and Design*, (A. Finkelstein, M.J. Tauber and R. Traunmuller, Eds.), North-Holland, 1990

[29] A. Motro, Constructing queries from tokens, *Proc. ACM-SIGMOD '86*, pp.120-131, 1986

[30] G. H. Sockut, L. M. Burns, A. Malhotra, K. Y. Whang, GRAQULA: A graphical query language for entity-relationship or relational databases, in *Data & Knowledge Engineering*, **11**, 1993, 171-202

[31] A. Albano, L. Alfò, S. Coluccini, R. Orsini, An Overview of Sidereus, a Graphical Database Schema Editor for Galileo, in *Advances in Database Technology - EDBT '88*, (J. W. Schmidt, S. Ceri and M. Missikof, Eds.), pp. 567-571, Springer-Verlag, 1988

[32] J. Rumbaugh , M. Blaha , W. Premerlani , F. Eddy , W. Lorensen , *Object-Oriented Modeling and Design*, Prentice-Hall, 1991

[33] C. Parent and S. Spaccapietra, An Algebra for a General Entity-Relationship Model, *IEEE Transactions On Software Engineering*, **11**(7), 1985, 634-643

[34] C. Parent, H. Rolin, K. Yétongnon, S. Spaccapietra, An ER Calculus for the Entity-Relationship Complex Model, in Entity-Relationship Approach to Database Design and Querying, (F. Lochovsky, Ed.), North-Holland, 1990

[35] S. Spaccapietra, C. Parent, K. Yétongnon, M. S. Abaidi, Generalizations: A Formal and Flexible Approach, in *Management of Data*, pp. 100-117, Tata McGraw-Hill, 1989

[36] Y. Dennebouy, *Un langage visuel pour la manipulation de données*, Ph Dissertation No. 1182, Swiss Federal Institute of Technology, Lausanne